Parser Combinators

We've discussed parsing lightly until this point. We will now dig down into the algorithmic details of parsing.

Before we start, you should know that there is a wealth of literature on parsing. For practical reasons, it was one of the earliest problems attacked by computer scientists. As a result, exploring this topic on your own can be a little daunting, as a typical description of parsing goes deep into the weeds about grammar classes, computational complexity, and so on. Compounding this, many computer scientists like to say offhandedly that parsing "is a solved problem," which is only true in the shallowest sense. Even with nice formal models from theoretical computer science, building a real-world parser remains something of an art.

Instead, we will look at parsing from a functional standpoint. A *parser* is a program that reads in a string as input and, if the input is a valid sentence in a grammar, (1) it emits a result, otherwise it (2) fails. This very simple definition allows us to construct a parser in a simple, recursive manner, using little building blocks. We call these building blocks *parser combinators*.

Why do we need parsers?

If you've never built a parser before, its role may not be obvious, so I will state it here clearly. In computer science, we use parsers to transform serial data (e.g., a string) into structured data (e.g., a tree). When building a programming language, the first thing we need to do is to convert a string (a program) into our preferred representation of a computer program, which is a tree. More precisely, that tree is an *abstract syntax tree* (AST).

An AST is a tree where the interior nodes are operations and the leaf nodes store data. Why do we want this representation? Because, in this form, evaluating a program boils down to a traversal of the tree. For example, in the form of program evaluation we call *interpretation*, we determine the "output" of a program by essentially performing a depth-first, post-order traversal of the tree, combining data from the leaves with op-

erations in the nodes. The output is the final value computed when we are done traversing the root node. In the form of program evaluation we call *compilation*, we also traverse the AST, but instead, we emit machine instructions as we go, converting each step of the interpreter into a sequence of instructions for a machine.

Parsers are used in many more places, from data storage to network protocols. You will probably encounter a few in your professional life. It suffices to say that we need them in the design of programming languages because they form the basis for building user interfaces for *humans*.

Parser Combinators

Before we dig into the technical details of how parser combinators work, let me try to develop an intuition as to what they do. There are two essential ideas.

The first essential idea regarding parser combinators is to build "big" parsing functions out of "little" parsing functions. And when I say *function*, I mean the simplest kind of function you can have: a *combinator*. A *combinator* is a function of *only bound variables*. In other words, this is a combinator:

```
let add a b = a + b
```

but this is *not* a combinator:

```
let add a = a + b
```

because `b` is a free variable. Where does `b` come from? It comes from the environment somewhere, and its precise meaning depends on the *scope rules* for your language. Therefore, a combinator is a function that can be understood without needing any context. It's a simple function without any tricks up its sleeve. A *parser combintor* is therefore a simple function that does parsing.

But how do we make "big" parsers out of "little" parsers? The second essential idea is that some parser combinators function as "glue." We call such "glue" functions *combining forms*.

Armed with "little" parsers and "glue," we can make "big" parsers of great sophistication that don't seem complex. Parser combinators are exactly the kind of big, ugly problem that becomes easy (or at least manageable) when you employ a functional programming approach.

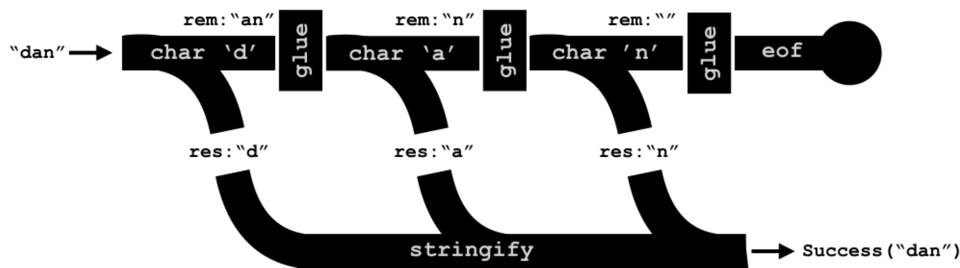
Figure 21: As a toddler, I once flushed a Kewpie doll down a toilet. When my father asked me where my doll was, I pointed at the toilet and said "In there." I have no memory of this incident, but my father, who had to disassemble the house's plumbing to find the doll, has yet to forgive me.

Metaphor

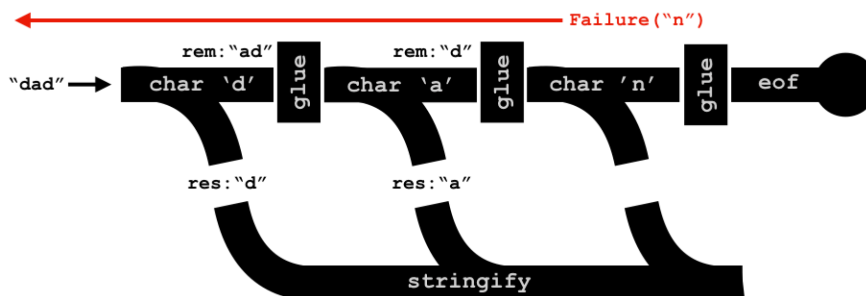
When I work with parser combinators, I like to keep a simple metaphor in my head: plumbing. The function of a pipe is to carry liquid from one point to another. As you're installing the plumbing in your house, you're only tangentially thinking about water (or sewage); you're thinking about how the shapes of the pipes fit together. Sometimes you join multiple pipes. Sometimes you split them. When you put the right stuff in the pipes, they do their jobs, moving liquid from one place to another. When you put stuff in pipes that you shouldn't, they get clogged and back up (Figure 21).

A combinator is like a pipe. It takes an input `string`, the `string` we are parsing. Depending on what the pipe does, it usually outputs a `string` of some form; that `string` represents the remainder of the input. But combinators may also connect to other combinators; the output of one combinator is fed into the input of the other.

When all goes well, and you flush the appropriate stuff down your parser pipes, you are able to parse input successfully. The parser shown here is designed to parse `"dan"`, and when given the input `"dan"`, it works just fine.



But when you flush the wrong stuff down the parser pipes, it backs up, and you get a failure.



Formal Definitions

Let's get a little more formal in our definition of parser combinators.

I said that "a parser is a program that reads in a string as input and, if the input is a valid sentence in a grammar, (1) it emits a result, otherwise it (2) fails." Let's start with the easy part, shall we? What is "input"?

Input

Here's a simple working definition.

```
type Input = string
```

We will revisit this definition later, but it's something to build on.

Success and Failure

What does it mean for a parser to "succeed" or "fail"?

You might be tempted to say that this means that a parser simply returns a `bool`, and if you were a theoretician studying grammars, that might be sufficient. As a practical matter, we usually expect parsers to return structured data, so we need something a little more nuanced. How about the following ML data structure?

```
type Outcome<'a> =
| Success of result: 'a
| Failure
```

We use `'a` because we might want to return any kind of data.

That's pretty close to what we want, but it's not perfect. The reason is that we want to be able to *combine* little parsers into big parsers. So one way to attack the problem of parsing is to think up a small set of primitive parsers that we can glue together that make more complicated parsers. Each parser then, takes a little nibble at the input and hands the *rest of the input*, the `remainder`, off to the next parser. So let's expand our definition:

```
type Outcome<'a> =
| Success of result: 'a * remainder: Input
| Failure
```

As a practical matter, we also add a small amount of extra debugging information to `Failure`: the position in the string that the failure occurred, and which parser failed.

```
type Outcome<'a> =
| Success of result: 'a * remainder: Input
| Failure of fail_pos: int * rule: string
```

That's good enough for now.

Parser

Now we can construct an elegant definition of a parser.

```
type Parser<'a> = Input -> Outcome<'a>
```

This definition says is that a parser is a function from input to an outcome, either success or failure. On success, we communicate back a result and the remaining portion of the input.

Primitive parsers

You may be surprised to hear that this is enough to start building primitive parsers. The two most primitive are parsers that either succeed no matter what or fail no matter what. We call them `presult` and `pzero`, respectively.

```
let presult(a: 'a)(i: Input) : Outcome<'a> = Success(a,i)

let pzero(i: Input) : Outcome<'a> = Failure(0, "pzero")
```

`presult` takes a return value (`'a`) and an input and returns success. `pzero` just returns failure.

Now, because both of these functions are written using *curried* arguments, they have an interesting and very useful property. If you call them without their last argument, the input, they return a `Parser<'a>`. I clearly remember the first time I learned this fact because my brain melted out through my ears. Maybe you're smarter than I am. But let's pop these definitions in `dotnet fsi` and play with them a bit just to be sure that we're on the same page.

```
> let presult(a: 'a)(i: Input) : Outcome<'a> = Success(a,i);;
val presult : a:'a -> i:Input -> Outcome<'a>

> let pzero(i: Input) : Outcome<'a> = Failure(0, "pzero");;
val pzero : i:Input -> Outcome<'a>

> presult;;
val it : ('a -> Input -> Outcome<'a>)

> presult "hi";;
val it : (Input -> Outcome<string>) = <fun:it@10-1>

> let p : Parser<string> = presult "hi";;
val p : Parser<string>
```

```
> pzero;;
val it : (Input -> Outcome<'a>)
```

```
> let p : Parser<'a> = pzero;;
val p : Parser<'a>
```

There's no magic here. Partially applying `presult` to `"hi"` returns a parser. `pzero` already is a parser.

OK, one more primitive parser. This is where the magic begins.

```
let pitem(i: Input) : Outcome<char> =
    if i = "" then
        Failure(0, "pitem")
    else
        Success (i.[0], i.[1..])
```

The `pitem` parser attempts to read in *one character*. Notice that the type of the `Outcome` is `char`. If it can read one character, then it returns that one character as the `result` (`i.[0]`) part of the `Success` value, putting the rest of the string (`i.[1..]`) in the `remainder` part. Otherwise, it fails.

Combining forms

Ok, we have three primitive parsers now. How do we "glue" them together? All combining forms are based on one idea, called "bind".

```
let pbind(p: Parser<'a>)(f: 'a -> Parser<'b>)(i: Input) : Outcome<'b> =
    match p i with
    | Success(a,i') -> f a i'
    | Failure(pos,rule) -> Failure(pos,rule)
```

Notice that we are prefixing all of our parser functions with the letter `p`. This is just to make it clear which functions belong to the primitive parsing library. You can name your functions whatever you want.

`pbind` takes a `Parser<'a>`, `p`, and a function `f` from `'a` to a new `Parser<'b>`, and returns a new `Parser<'b>`. Why do I say that it "returns a new parser" when that's not precisely what the definition says? Well, look carefully in the REPL; it *does* say that. You're just not accustomed to seeing it yet. Here's an example of me partially-applying `pbind`.

```
> let p : Parser<char> = pbind pitem (fun c -> pitem);;
val p : Parser<char>
```

The key bit is that I left off the `Input`, `i`. Remember how I said that all parser combinators take `Input` as their last argument, and, if you leave

it off, they're parsers? This is what I meant. That returned parser does the following:

1. Attempt to parse `Input i` with `p`.
2. On success, run `f` on the result of the successful parse, yielding a new parser, `p2`.
3. Run `p2` on the remainder of the first parse.
4. If `p2` is successful, return the outcome of the second parse, otherwise fail.

If you're like me, you might be thinking "OK, I can see *that* you can glue parsers together, but how is this useful?" Great question. I think the most obvious answer is: don't we want to be able to parse more than one character? So let's see how we can achieve that using what we know.

Parsing in sequence

Let's construct a new combining form called `pseq`. We'll use this to parse *two* characters.

```
let pseq(p1: Parser<'a>)(p2: Parser<'b>)(f: 'a*'b -> 'c) : Parser<'c> =
    pbind p1 (fun a ->
        pbind p2 (fun b ->
            presult (f (a,b))
        )
    )
```

The `pseq` parser is a "combining" function. It takes two parsers, `p1` and `p2`, and runs them, one after the other, returning the result as a pair of elements. Note that we use `pbind` in this definition, and `presult` finally makes an appearance. We first bind `p1` to a function that takes `p1`'s result as input and then binds `p2` to a function that takes `p2`'s result, which is then handed to the `presult` parser, which takes both results and runs function `f` on them. This may seem sort of abstract to you, but if you work it out on paper, it's not so bad. It captures everything we need to say in order to parse two things in sequence.

Now, notice, that this definition does not take an `Input`. Or does it? Actually, it does! But we were able to leave it off. Why? Because `p1`, `p2`, `pbind`, and `presult` also all take an `i`, and since we only ever partially apply those functions, we are always "passing the buck" to the next function in the chain. Since the entire body of the `pseq` function punts on handling input, *even though its components must handle input*, it means that `pseq` itself must handle input. In fact, that's what the return type says: `Parser<'c>`.

Part of the reason why this sort of melted my brain the first time I saw it is that I kept thinking: but why don't you just handle the input?

Wouldn't the definition be clearer? The PL theorist who taught me combinators thought the answer to this question was obvious ("NO!") so he didn't spend much time on it. As a result, it took me a little time to appreciate how much simpler partial application can make a program. The gain in simplicity is especially profound when it is the case that all of your functions pass around the same parameter (in our case, `Input`).

This fact sheds light on the popularity of a another model for programming: object oriented programming. In that model, you pass around all kinds of parameters implicitly—you just stick that data inside an object and pass the object around instead. So it solves the same problem, but using a different mechanism. But unlike functional programming, where you are *forced* to think about all of the data you need all of the time, we sometimes forget about the data we stick in objects. In particular, we forget that we need to update it, leading to bugs. Functional code forces us think about that data. The tradeoff is that we never have stale values floating around in objects. Personally, functional programming forced me to stop being lazy with objects, and the benefits—fewer bugs—became immediately clear to me.

OK, enough chit-chat. Let's use `pseq` to actually parse two characters.

```
let ptwo : Parser<string> =
  pseq pitem pitem (fun (c1,c2) -> c1.ToString() + c2.ToString())
```

So we just constructed a parser that parses two characters, and then takes the pair of characters, converts them to strings (remember a `char` is not a `string` and in F# we have to explicitly convert them), and concatenates them, returning a string. Let's try it.

```
> ptwo "hello world";;
val it : Outcome<string> = Success ("he","llo world")
```

Cool, huh? Watch what happens when the input does not have two characters left.

```
> ptwo "h";;
val it: Outcome<string> = Failure (0, "pitem")
```

Because we built up our parsers simply and from first principles, the combined parser does the right thing.

End of file

There's one more essential parser that we need to specify, and it requires that we change our definition of input a little. It is often necessary, for example, in your "top level" parser, to be able to state "only succeed if you've parsed all of the input." In other words, we need to check that

we've reached the end of the input string.

Unfortunately, nowhere in our definition of `Input` do we maintain a notion of "the end". We probably should. Let's modify our definition of `Input` just a little. [57]

```
type Input = string * bool
```

Now `Input` is a pair of `string` and `bool`. The `bool` represents whether we've found the end. This is useful because often a parser definition, which is composed of many little parsers, slices and dices the input into many pieces, and those pieces themselves are sliced and diced. Without tracking the "real end" of the string, we might be tempted to think that "the end" was merely the end of the input string. But if we've cut the string in half somewhere, that most definitely will not be the case. There's only *one end*!

This affects the definition of `pitem` above, but not by much. In fact, it doesn't change at all how we use them, just whether we can actually test for EOF. Here's a definition of an EOF parser:

```
let peof(i: Input) : Outcome<bool> =
    match pitem i with
    | Failure(pos, rule) ->
        if snd i = true then
            Success(true, i)
        else
            Failure(pos, rule)
    | Success(,) -> Failure((position i), "peof")
```

First, `peof` tries to get a character, and if that fails *and we're at the real end of the string*, succeed. Otherwise, fail.

Here's a little function that we can call on our input `string` to turn it into an `Input` so that the user does not have to think about how to set up an `Input` the first time.

```
let prepare(input: string) : Input = input, true
```

[57] Note that the combinator library that comes with your starter code, `Combinator.fs`, has an even fancier definition for `Input` for efficiency. The basic idea is the same, but note that the definition is slightly different. Have a look if you are curious. I try to make our libraries easy to read.

A zillion more little parsers

Hopefully now you have the basic idea. You can make and combine parsers from other parsers. That combined parser can be called using string input, and it returns what you ask. At each step, you provide a function `f` that says exactly how to build the data structure that you return in the end (e.g., "concatenate two characters into a string"). It can be whatever you want.

In this section, I am going to tell you about a collection of other useful parsers. I am not going to belabor their definitions, since you can just look through the code and understand them if you need to. In many cases, you will not need to. This, of course, is also not an exhaustive list of parsers. Like I said, they build pretty much anything you want. The set below is just a subset convenient to use for this course.

<table>
<tr><th>Parser</th><th>Type</th><th>Description</th><th>Example</th></tr>
<tr><td>presult</td><td>'a -> Input -> Outcome<'a></td><td>Takes a result value 'a and an input and returns Success.</td><td></td></tr>
<tr><td>pzero</td><td>Input -> Outcome<'a></td><td>Takes an input and returns Failure.</td><td></td></tr>
<tr><td>pitem</td><td>Input -> Outcome<char></td><td>Reads a single character.</td><td></td></tr>
<tr><td>peof</td><td>Input -> Outcome<bool></td><td>Takes an input and returns true if and only if there is no more input left to consume.</td><td></td></tr>
<tr><td>pbind</td><td>p:Parser<'a> -> f:('a -> Parser<'b>) -> Input -> Outcome<'b></td><td>Form for combining a parser p in an arbitrary way with another parser using a function f.</td><td></td></tr>
<tr><td>pseq</td><td>p1:Parser<'a> -> p2:Parser<'b> -> f:('a * 'b -> 'c) -> Parser<'c></td><td>Combine two parsers p1 and p2 in sequence, and combine their results using a function f.</td><td>pseq pitem pitem (fun (a,b) -> (a,b) parses two characters and returns them as a 2-tuple.</td></tr>
<tr><td>psat</td><td>f:(char -> bool) -> Parser<char></td><td>Read a character, and if it satisfies the predicate f, successfully return the character.</td><td>psat (fun c -> c = 'z') parses the character z</td></tr>
<tr><td>pchar</td><td>c:char -> Parser<char></td><td>Read a character, and if it is the same as the given character c, successfully return it.</td><td>pchar 'z' parses the character z</td></tr>
<tr><td>pletter</td><td>Parser<char></td><td>Reads a character and returns successfully if the character is alphabetic.</td><td>pletter parses any alphabetic letter.</td></tr>
<tr><td>pupper</td><td>Parser<char></td><td>Reads a character and returns successfully if the character is uppercase alphabetic.</td><td>pupper parses any uppercase alphabetic letter.</td></tr>
<tr><td>pdigit</td><td>Parser<char></td><td>Reads a character and returns successfully if the character is numeric.</td><td>pdigit parses any numeral.</td></tr>
<tr><td><|></td><td>p1:Parser<'a> -> p2:Parser<'a> -> string * bool -> Outcome<'a></td><td>Ordered choice. Note that this is an infix combinator, for readability. First tries the parser p1, and if that fails, it backtracks the input and tries parser p2. The first parser to succeed returns the result. If both parsers fail, choice fails.</td><td>(pchar 'a') <|> (pchar 'b') parses either the character a or the character b.</td></tr>
<tr><td>|>></td><td>p:Parser<'a> -> f:('a -> 'b) -> Parser<'b></td><td>Function application. Applies the function f to the result of p if p is successful.</td><td>pdigit |>> (fun c -> int (string c)) converts a numeric character into an integer.</td></tr>
</table>

Parser	Type	Description	Example
pfresult	p:Parser<'a> -> x:'b -> Parser<'b>	Run parser p and if successful, return result x. This basically ignores the output of p.	pfresult (pchar 'a') 'b' returns the character b when it finds a.
pmany0	p:Parser<'a> -> string * bool -> Outcome<'a list>	Parse zero or more occurrences of p in sequence, stopping when p fails. Note that this parser never fails!	pmany0 pletter parses a sequence of letters, stopping when no more letters can be found.
pmany1	p:Parser<'a> -> Parser<'a list>	Parse one or more occurrences of p in sequence, stopping when p fails. To succeed, p must succeed *at least once*.	pmany1 pletter parses a sequence of letters of length 1 or more.
pws0	Parser<char list>	Parses a sequence of zero or more whitespace characters.	pws0
pws1	Parser<char list>	Parses a sequence of one or more whitespace characters.	pws1
pnl	Parser<string>	Parses a newline. Returns a string instead of a char because newlines are actually two characters on Windows machines.	pnl
pstr	s:string -> Parser<string>	Parses the string literal s.	pstr "helloworld" parses helloworld and only helloworld.
pleft	p1:Parser<'a> -> p2:Parser<'b> -> Parser<'a>	Parses p1 and p2 in sequence, returning *only* the result of p1. Discards the result of p2.	pleft (pchar 'a') (pchar 'b') parses ab but only returns a.
pright	p1:Parser<'a> -> p2:Parser<'b> -> Parser<'a>	Parses p1 and p2 in sequence, returning *only* the result of p2. Discards the result of p1.	pright (pchar 'a') (pchar 'b') parses ab but only returns b.
pbetween	popen:Parser<'a> -> pclose:Parser<'b> -> p:Parser<'c> -> Parser<'c>	Parses p *in between* parsers popen and pclose. Discards the resuls of popen and pclose.	pbetween (pchar '(') (pchar ')') (pmany0 pletter) returns abc when given (abc).
<!>	p:Parser<'a> -> label:string -> string * bool -> Outcome<'a>	Debug parser. This parser applies p and, as a side effect, prints some diagnostic information given a label. Very useful for figuring out why a parser succeeds or fails on a given input.	pletter <!> "letter"
stringify	cs:char list -> string	Convert the char list called cs into a string.	stringify ['h';'e';'l';'l';'o'] returns hello

An example: parsing English sentences

Lets's build a small parser for parsing well-formed English sentences. As you will see, it's not hard to find the limitations of this parser. But after we build this together, you should have a good idea about how you could extend it to parse more complex sentences.

First, what is an "English sentence"? Here's some BNF:

```
<sentence>    ::= <upperword> (<ws> <word>)* <period>
<upperword>   ::= <upperletter> (<letter>)*
<word>        ::= (<letter>)+
<upperletter> ::= 'A' | 'B' | ... | 'Y' | 'Z'
<lowerletter> ::= 'a' | 'b' | ... | 'y' | 'z'
<letter>      ::= <upperletter> | <lowerletter>
<ws>          ::= ' ' | 'n' | 't' | "rn"
<period>      ::= '.'
```

Let's further stipulate that the "structure" that I want to return from my parser is a list of the words in the sentence. A `string list` should work nicely. I would also like to know whether the parse succeeded or failed, so let's wrap our `string list` in an option type[58]. So our end result will be a function like:

[58] https://docs.microsoft.com/en-us/dotnet/fsharp/language-reference/options

```
let parse(input: string) : string list option =
    ... whatever ...
```

When building a parser using combinators, you can either start at the top of your grammar and work your way down or you can start at the bottom and work your way up. I'm sort of a bottom-up thinker, so we'll start with the simplest parts; the ones that parse terminals.

Period has a simple grammar rule with only one terminal. Should be easy.

```
let period = pchar '.'
```

This is hopefully self-explanatory.

Whitespace, as it turns out, is built-in. Let's say, for now, that `pws1` is what we want.

OK, arbitrary letters. Again, we already have a parser for this called `pletter` that parses both uppercase and lowercase letters. We also have parsers for uppercase only (`pupper`) and lowercase onle (`plower`).

How about words? Our word production says:

```
<word> ::= (<letter>)+
```

What does that mean? We're using an "extended" form of BNF called EBNF here that lets us write repetition more concisely. + means "at least one". So what this says is "at least one `<letter>`". We can unroll this if we want into regular BNF.

```
<word> ::= <letter> <word>
        |  <letter>
```

So a `<word>` really is a recursive definition that requires at least one `<letter>`. Fortunately for us, we don't have to think too hard about repetition, because there's a parser combinator that means "one or more" called `pmany1`. So a word is:

```
let word = pmany1 pletter
```

Which is great and all, but *almost* what we want. Remember how I said that we wanted a "list of words" back and I said that this translated into a `string list`? Well, what does `pmany1 pletter` actually return?

```
> let word = pmany1 pletter
val word : Parser<char list>
```

It actually returns a list of characters. Although I think we all can agree that a list of characters is pretty much a string, we have to actually convert one into the other to keep F# happy. We do that using the `|>>` combinator.

```
> let word = pmany1 pletter |>> (fun cs -> System.String.Join("", cs))
val word : Parser<string>
```

That's better.

Because we often translate lists of characters into strings, I've provided the `stringify` function that does this for you. So we can rewrite `word` a little more simply.

```
let word = pmany1 pletter |>> stringify
```

Let's test our work up until this point. Remember that we need to call `prepare` on our input `string` before we can give it to our parsers.

```
> word (prepare "foobar");;
val it : Outcome<string> = Success ("foobar",("", true))
```

That looks promising. How does it handle a space in the middle?

```
> word (prepare "foo bar");;
val it : Outcome<string> = Success ("foo",(" bar", true))
```

Notice that it succeeds, but only returns `"foo"`. `"bar"` is left in the remainder. This makes sense because `word` doesn't know anything about spaces.

```
> word (prepare " foo bar");;
val it : Outcome<string> = Failure
```

This also looks good. There are words in the string, but the string starts with a space. Again, `word` doesn't know anything about spaces so it fails.

How about words that start with an uppercase letter?

```
<upperword> ::= <upperletter> (<letter>)*
```

Again, this is EBNF. The * operator means "zero or more". So an uppercase word must be at least one uppercase letter followed by zero or more letters of any case. Unrolled into regular BNF:

```
<upperword> ::= <upperletter> <word>
              |  <upperletter>
```

As before, thinking about this recursively is a useful exercise, but we have a parser that makes our lives easier. `pmany0` parses zero or more occurrences of a parser. How do we parse first an uppercase letter and then zero or more letters of any case? Anytime your parsing logic is of the form "first do this then do that" you're talking about sequences.

```
let upperword = pseq pupper (pmany0 pletter)
```

As before, we want to get back a `string`, but what this gives us back is kind of a mess. `pmany0 pletter` returns a `char list`, `pseq` returns a tuple, and `pupper` returns a char, so what we get is a `char * char list`. Fortunately, `pseq` also expects a function that lets us sort it all out. We want a `string`.

```
> let upperword = pseq pupper (pmany0 pletter) (fun (x,xs) -> stringify (x::xs));;
val upperword : Parser<string>
```

Now it does what we want!

OK, where are we? We can parse uppercase words and other words. What is the form of a sentence? From our BNF above, it really has

two pieces. Let's switch from working bottom-up to working top-down. Hopefully we can meet in the middle somewhere.

```
<sentence> ::= <upperword> (<ws> <word>)* <period>
```

There's a *prefix*, which is the first uppercase word and a middle part, which is spaces and words. Then there's a *suffix*, which is the period. When you have a complicated production rule, thinking in terms of prefixes and suffixes helps a lot. Note that we could have divided this in many different ways. Let's make a top-level `sentence` parser with these parts and then flesh each piece out.

```
let sentence = pleft prefix period
```

`pleft` applies two parsers but only returns the result of the one on the left. So `sentence` just returns the result of `prefix`, which makes sense because we don't actually care about putting a period in our list of words.

`prefix` also has two parts: an uppercase word and then zero or more whitespace-separated words.

```
let prefix = pseq upperword words0 (fun (w,ws) -> w::ws)
```

We are calling "zero or more whitespace-separated words," `words0`. OK, so clearly `upperword` returns a string. And hopefully, we can build `words0` so that it returns a `string list`. If that's what we're getting back, then combining them should be simple: just *cons* the uppercase word to the list of words from `words0`. Fortunately, `pseq` wants a function that asks us how to combine its two pieces, so we tell it to combine using *cons*.

Let's define `words0` now. So we want zero or more words prefixed by whitespace.

```
let words0 = pmany0 (pright pws1 word)
```

`pright` is like `pleft` except that it returns the result from the parser on the right, in this case, a `word`. Without any more work, this already does what we want, see?

```
> let words0 = pmany0 (pright pws1 word);;
val words0 : (Input -> Outcome<string list>)
```

which, of course, is a `Parser<string list>`.

We're almost done! We just have to define a top-level parser now.

Out of habit, I always call this top-level parser `grammar`. `grammar` really only does one thing: it calls our parser and makes sure that we've parsed *all* of the input. Remember that many parsers (like `word`) will happily nibble off only a part of the input and leave the rest behind. To ensure that all the input is consumed, we make sure that the only thing left is EOF.

```
let grammar = pleft sentence peof
```

Since we don't really care what `peof` returns—just that it's successful—we use `pleft` with our `sentence` parser.

Here is the complete program along with a little `main` function so that you can try it out using `dotnet run`.

```
open Combinator

let period = (pchar '.')
let word = pfun (pmany1 pletter) (fun cs -> stringify cs)
let upperword = pseq pupper (pmany0 pletter) (fun (x,xs) -> stringify (x::xs))
let words0 = pmany0 (pright pws1 word)
let prefix = pseq upperword words0 (fun (w,ws) -> w::ws)
let sentence = pleft prefix period
let grammar = pleft sentence peof

let parse input : string list option =
    match grammar (prepare input) with
    | Success(ws,_) -> Some ws
    | Failure(_,_) -> None

[<EntryPoint>]
let main argv =
    if argv.Length <> 1 then
        printfn "Usage: dotnet run <sentence>"
        exit 1
    match parse argv.[0] with
    | Some ws -> printfn "Sentence: %A" ws
    | None    -> printfn "Invalid sentence."
    0
```

The parsers I describe above are available in a module called `Combinator.fs` which is available on the course website.

Let's run our program.

```
$ dotnet run "This is a sentence."
Sentence: ["This"; "is"; "a"; "sentence"]
```

While you can go around sticking `printfn` statements into your combinator code when things don't go as planned, there's a much better way to debug: the debug parser, `<!>`. Here's a version of the same program but this time decorated with debug parsers.

```
open Combinator

let period = (pchar '.') <!> "period"
let word = pfun (pmany1 pletter) (fun cs -> stringify cs) <!> "word"
let upperword = pseq pupper (pmany0 pletter) (fun (x,xs) -> stringify (x::xs)) <!> "upperword"
let words0 = pmany0 (pright pws1 word) <!> "words0"
let prefix = pseq upperword words0 (fun (w,ws) -> w::ws) <!> "sprefix"
let sentence = pleft prefix period <!> "sentence"
let grammar = pleft sentence peof <!> "grammar"

let parse input : string list option =
    match grammar (debug input) with
    | Success(ws,_) -> Some ws
    | Failure(_,_) -> None

[<EntryPoint>]
let main argv =
    if argv.Length <> 1 then
        printfn "Usage: dotnet run <sentence>"
        exit 1
    match parse argv.[0] with
    | Some ws -> printfn "Sentence: %A" ws
    | None    -> printfn "Invalid sentence."
    0
```

Observe that we changed `prepare input` in our `main` method to `debug input`. The difference is that `debug` sets an internal debugging flag to `true` whereas `prepare` sets it to `false`. The two functions are otherwise the same. When a debug parser, always written like `<!> "rule"`, encounters a debugging flag set to `true`, it prints diagnostic information, including `rule`, to the terminal.

Let's run this program.

```
$ dotnet run "This is a sentence."
[attempting: grammar on "This is a sentence.", next char: 0x54]
[attempting: sentence on "This is a sentence.", next char: 0x54]
[attempting: sprefix on "This is a sentence.", next char: 0x54]
[attempting: upperword on "This is a sentence.", next char: 0x54]
[success: upperword, consumed: "This", remaining: " is a sentence.", next char: 0x20]
[attempting: words0 on "This is a sentence.", next char: 0x54]
[attempting: word on "This is a sentence.", next char: 0x54]
[success: word, consumed: "is", remaining: " a sentence.", next char: 0x20]
[attempting: word on "This is a sentence.", next char: 0x54]
[success: word, consumed: "a", remaining: " sentence.", next char: 0x20]
[attempting: word on "This is a sentence.", next char: 0x54]
[success: word, consumed: "sentence", remaining: ".", next char: 0x2e]
[success: words0, consumed: " is a sentence", remaining: ".", next char: 0x2e]
[success: sprefix, consumed: "This is a sentence", remaining: ".", next char: 0x2e]
[attempting: period on "This is a sentence.", next char: 0x54]
[success: period, consumed: ".", remaining: "", next char: EOF]
[success: sentence, consumed: "This is a sentence.", remaining: "", next char: EOF]
[success: grammar, consumed: "This is a sentence.", remaining: "", next char: EOF]
Sentence: ["This"; "is"; "a"; "sentence"]
```

With this latter version, when things go wrong, we can see why.

```
$ dotnet run "This is a sentence"
[attempting: grammar on "This is a sentence", next char: 0x54]
[attempting: sentence on "This is a sentence", next char: 0x54]
[attempting: sprefix on "This is a sentence", next char: 0x54]
[attempting: upperword on "This is a sentence", next char: 0x54]
[success: upperword, consumed: "This", remaining: " is a sentence", next char: 0x20]
[attempting: words0 on "This is a sentence", next char: 0x54]
[attempting: word on "This is a sentence", next char: 0x54]
[success: word, consumed: "is", remaining: " a sentence", next char: 0x20]
[attempting: word on "This is a sentence", next char: 0x54]
[success: word, consumed: "a", remaining: " sentence", next char: 0x20]
[attempting: word on "This is a sentence", next char: 0x54]
[success: word, consumed: "sentence", remaining: "", next char: EOF]
[success: words0, consumed: " is a sentence", remaining: "", next char: EOF]
[success: sprefix, consumed: "This is a sentence", remaining: "", next char: EOF]
[attempting: period on "This is a sentence", next char: EOF]
[failure at pos 18 in rule [pchar '.']: period, remaining input: "", next char: EOF]
[failure at pos 18 in rule [pchar '.']: sentence, remaining input: "", next char: EOF]
[failure at pos 18 in rule [pchar '.']: grammar, remaining input: "", next char: EOF]
Invalid sentence.
```

Oops! It looks like we forgot the period.

Performance

One thing you may notice while playing with combinators is that the performance is not always stellar. There are a few reasons.

First, this library is not optimized in any way. It's designed as a teaching tool. If you want a commercial-grade combinator parsing library, you should look elsewhere. FParsec[59] is a good, open-source library that I use a lot.

[59] http://www.quanttec.com/fparsec/

Second, backtracking parsers are expensive, because when they fail, other alternatives are explored. For example, if you peek at the imlementation for the choice combinator, `<|>`, you will see that it does just that. When producing a commercial-grade parser, you will want to invest some time in optimizing your code. These optimizations are largely outside the scope of this class.

Finally, there is a cost to using higher-order functions, although the F# compiler does do a respectable job about optimizing away obvious inefficiencies. Still, *hand-written parsers*, not using parser libraries like combinators, will always be faster, especially in languages like C. Consequently, some of the fastest parsers are written in C. This is sometimes essential in domains where performance is critical, as in code for network protocols.

Parsing theory is good, but hard to apply in practice

Before I go, I want to revisit my comments about parsing theory. As I stated before, there is a ton of research on parsing. In fact, some parsers can be *generated automatically* using tools called *parser generators*. These generated parsers can even be blazingly fast because they can be designed to emit C code that is basically one big `switch` statement. This is called a *table-driven parser*. The chief difficulty with using a parser generator, however, is that you need to know the formal grammar class of your language. Is your language "deterministic context-free?" Great! Use an LR parser. Is it a regular language? Great! Use regular expressions.

In practice, it is often difficult to know for sure what class your language belongs to until you try to generate a parser. And even though

we can generate parsers, this does not mean that it is "no work" to generate the specification for the parser generator. In my experience, you are often deep into a parser design when you stumble across a syntatic construct that cannot be parsed using an LR parser. Oh crap! Now what?! Change your syntax or throw out the entire spec and start over again with another parser generator? People who do this more often than I do probably have a better intuition for which parser generators are good for which jobs. It suffices to say that when I discovered combinators, I entirely stopped using parser generators and never looked back.

Anyway, the fact that the "best" (or really, "best known") parsing algorithm can be chosen based on the grammar class of your language is why theoretical computer scientists call parsing a "solved problem." Like many other things in computer science, though, the proof is in the pudding.